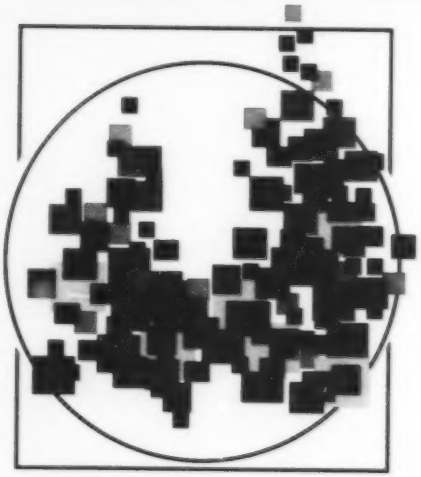

CENTRE FOR
SYSTEMS
SCIENCE

Simon Fraser University, Burnaby, B.C., Canada V5A 1S6 / (604) 291-4704

Reduction Reconsidered: Further Investigation of a Mechanism for Searching for Regularity in Data

Howard J. Hamilton

CSS-IS TR 89-09

December 1989

Reduction Reconsidered: Further Investigation of a Mechanism for Searching for Regularity in Data

Howard J. Hamilton
18 December 1989

Abstract

Recently, Wu proposed a method called *Reduction* for searching for regularities in experimental data [Wu 88]. Reduction was described as a polynomial-time algorithm sufficiently powerful to make significant scientific discoveries in "uncultivated" domains. However, Reduction is not an efficient, general-purpose method for discovering regularities in data. *Basic Reduction*, the polynomial-time method proposed by Wu works correctly only for a subset of the cases for which he proposed it, and *Universal Reduction*, his more general method, requires exponential time. As well, Reduction requires the existence of a function that depends on all of the variables and yields a constant value for all data tuples, but this requirement is not fulfilled in most practical cases. Reduction makes unrealistic assumptions about the availability of data tuples: it assumes that data tuples can be provided whenever values are specified for a subset of the variables. Finally, Reduction's method of verifying a function may fail for any function whose decomposition includes a function whose inverse is many-to-one rather than one-to-one. A consideration of the limitations just described leads to a list of conditions which must be satisfied for Basic Reduction to be used for an application. As well, a more efficient algorithm for such applications is proposed.

Copyright © 1989 Howard J. Hamilton

This work was supported by a Simon Fraser University Graduate Fellowship, a B.C. Advanced Systems Foundation Graduate Research Scholarship, and Natural Sciences and Engineering Reseach Council of Canada Grant A4309.

92-05426/

Table of Contents

1. Introduction 1
2. Reduction Method 2
   2.1. Algorithm 2
   2.2. Complexity Analysis 6
      2.2.1. Complexity of the Outer Loops 7
      2.2.2. Cost of One Iteration of the *U*-Loop 7
      2.2.3. Summary of Complexity Analysis 9
3. Analysis of Assumptions 10
   3.1. Examples 10
      3.1.1. Type of Examples 10
      3.1.2. Format of Examples 10
      3.1.3. Presentation of Examples 11
      3.1.4. Number of Examples 12
   3.2. Hypotheses 14
      3.2.1. Target Functions 14
      3.2.2. Prototypes and Target Function Decomposition 16
      3.2.3. Number of Prototypes 17
      3.2.4. Choice of Prototypes 17
      3.2.5. Summary of Assumptions about the Hypothesis Space 18
   3.3. Method 18
      3.3.1. Obtaining Match Data 18
      3.3.2. Finding a Function 20
      3.3.3. Testing the Function 23
      3.3.4. Updating Variables 26
4. Improved Algorithm for Basic Reduction 27
5. Conclusions and Suggestions for Further Research 31
Acknowledgements 35
References 36

1. Introduction

Recently, Wu proposed a method called *Reduction* for searching for regularities in experimental data [Wu 88]. Reduction was described as a polynomial-time algorithm that is both more efficient than the methods used in BACON ([Langley 78] [Langley *et al* 83, 86]), ABACUS [Falkenhainer and Michalski 86], and COPER [Kokar 86] and sufficiently powerful to make significant scientific discoveries in uncultivated domains. By an "uncultivated domain", Wu appears to mean a scientific field where observational data are being accumulated and need to be analyzed for numerical relationships.

Although Wu's paper is a weak one, it represents an initial effort at attempting to incorporate numerical-analysis techniques into a program for automatic scientific discovery. Two weaknesses in previous efforts at automating scientific discovery have been the inability to handle errors in data and inefficiency in searching for regularities in numeric data. Incorporating numerical-analysis techniques seems to be a promising way of overcoming these weaknesses. Wu's paper focuses on the incorporation of numerical-analysis techniques to improve efficiency by searching for classes of relations between variables instead of individual relations. As a simple example, Reduction considers all functions of the form $f(x) = x^i$ at once rather than considering each of $f(x) = x$, $f(x) = x^2$, $f(x) = x^3$,... separately, as BACON does. Unfortunately, *Basic Reduction*, the polynomial-time method proposed by Wu, succeeds only for a restricted class of problems, and *Universal Reduction*, his more general method, requires exponential time. As well, for the class of problems for which Basic Reduction succeeds, a more efficient polynomial algorithm exists.

In this paper, Reduction is examined to identify the conditions required to permit its claimed efficiency and the corresponding limitations in its applicability. In Section 2, Reduction is briefly described and an algorithm is presented based on the description of the method given in [Wu 88]. Also, complexity analyses of Basic and Universal Reduction are given. In Section 3, unstated assumptions made in the design of Reduction are explicit., documented. In Section 4, the particular class of problems for which Basic Reduction is applicable is characterized, and a more efficient algorithm for such problems is proposed. Finally, in Section 5, conclusions and suggestions for further research are presented.

2. Reduction Method

Reduction is intended to solve an induction problem which may be called the *numeric-data analysis problem*. *Induction* is the process of forming a general concept on the basis of specific examples. An induction problem may defined in terms of an *example space* and a *hypothesis space*, where the hypotheses are possible descriptions of the examples. In numeric-data analysis, the examples are data tuples giving numeric values for a set of variables and the hypotheses are sets of mathematical equations and inequalities involving the variables. Reduction attempts to solve a restricted version of this problem in which each hypothesis is an equation between a numeric constant and a nontrivial function[1] on all the variables. Such an equation is called a *target equation*[2], and the included function is called a *target function*. Thus, Reduction attempts to find a nontrivial function on a set of variables that yields a constant value for all data tuples. For example, from a sequence of (x,y,z) data tuples including {(1,0,1), (1.2,0.44,1), (2,3,1), (2.5,5.25,1), (2,0.75,2), (3,2,2)},[3] Reduction finds the target function x^2-yz^2, which has a constant value of 1 for these data tuples.

2.1. Algorithm

The method is called "Reduction" because the task of discovering a target equation on n variables is reduced to discovering simpler equations on fewer variables. Given a set of n variables, the Reduction method first finds a *constant-valued function*, a nontrivial function on a subset of two or more of the variables that yields a constant value for all examined data tuples. Then the subset of the variables is replaced by a newly created *synthetic variable* defined to have the value given by applying the constant-valued function to the subset of variables. The process of finding a constant-valued function and replacing the relevant variables with a synthetic variable is repeated until only a single variable remains; this variable has a constant value for all examined data tuples. Beginning with the final variable, the synthetic variables are replaced by their definitions in terms of constant-valued functions until the target function is produced. Since the

---

[1] A nontrivial function on n variables cannot be redefined using a proper subset of the variables. Wu does not restrict the class of target functions to nontrivial functions; this decision leads to the unsatisfactory result that the target equation $f(x_1,\ldots,x_n) = 0$ satisfies his requirements for all problems, because the function f yields a constant value for all data tuples. Therefore, target functions are restricted to nontrivial functions in the present paper.

[2] Wu calls it a "final law" or an "original formula".

[3] As example data for x^2-yz^2, Wu gives {(1,0), (1.22,0.44), (2,3), (2.5,1.25)} with $z=1$ and {(2,3/4), (3,2)} with $z=2$, but the values 1.22 and 1.25 are incorrect and have been replaced by 1.2 and 5.25 in the data given here.

substitution is straightforward, it will not be considered further.

An outline of the Universal Reduction method is as follows:

Set *S* to the original set of variables.
while *S* contains more than one variable:
    for all subsets *U* of two or more variables in *S*, beginning with the smallest:
        If a constant-valued function on *U* can be found
        then Remove the variables in *U* from *S*;
            Create a new synthetic variable corresponding to *f* and add it to *S*;
            Restart the **for** loop with the new value for *S*.
    If no constant-valued function is found for any subset
    then the method fails.

The Universal Reduction method is easily seen to have exponential time complexity because in the worst case, all subsets of *S* (except those containing fewer than 2 elements) are examined. Basic Reduction, the polynomial-time restricted version of Universal Reduction, is obtained by restricting the size of the subsets to 2. A more detailed complexity analysis of the algorithms is postponed until Section 2.2.

Reduction also includes a method for finding a constant-valued function (if one exists) on a set of variables *U*. The method assumes that a set of parameterized functional forms (called *prototypes*) is provided. An outline of the method is as follows:

Obtain a set of data tuples *D* with a variety of values for the variables in *U* and constant values for the other variables.
for all prototypes *p*
    Attempt to assign values to the parameters of *p* such that the resulting function *f* matches all the data tuples in *D*;
    If a match is found
    then Obtain pairs of data tuples, with each pair having two different values for one of the variables not in *U* and a constant value for the other variables not in *U*;
        Verify that *f* is a constant-valued function on each of these pairs.

Algorithm Reduction1, an algorithmic formulation of the Universal Reduction method described in [Wu 88], is now presented, preceded by a table of program variables[4] and constants used in the algorithm. (An algorithm for the "Basic Reduction" method is obtained by restricting the value of *i* to 2.)

---

[4]These variables are called "program variables" to distinguish them from the "mathematical variables" referred to elsewhere in this paper.

Program Variables and Constants for Algorithm Reduction1	
n	The number of original variables.
V	The set of variables. Originally, $V = \{V_1,...,V_n\}$. Up to $n-1$ synthetic variables are created and added to V by Function Synthesize_Variable.
c	The number of synthetic variables that have been defined.
P_i	The set of parameterized functional forms (prototypes) on i variables.
P	The set of parameterized functional forms (prototypes). $P = \bigcup_{i=1}^{n}(P_i)$.
S	The set of variables remaining to be related together.
$OldS$	A copy of S before the current attempt at finding some related variables.
U	The set of variables that the algorithm is currently trying to relate together.
D	A set of data tuples to be used for attempted matches with prototypes.
$\#D$	The number of data tuples to be used for attempted matches with prototypes.
e	The current candidate for an equation on a subset of the variables.
f	The current candidate for a constant-valued function on a subset of the variables.

Algorithm Reduction1

(* Given a set V of n variables $\{V_1,...,V_n\}$ and access to data tuples providing values for these variables, this algorithm either finds and returns a target equation or returns **null** if no equation can be found. A set P of prototypes (i.e., parameterized functional forms) is also assumed to be available. The number of data tuples to be used for matching against prototypes is given by $\#D$. The notation $|S|$ denotes the number of elements in set S. *)

```
            c ← 0
            S ← V
S-loop:     while |S|>1
                OldS ← S
i-loop:         for i = 2,3,...,|S|
U-loop:             for all U ⊆ S such that |U| = i
                        D ← Get_Data(V,S − U,∅,#D)
p-loop:                 for all p ∈ P_i
                            {e,f} ← Match_Prototype(p,U,D)
                            if f ≠ null and Verify_Function(V,S − U,f)
                            then c ← c + 1
                                 Synthesize_Variable(V_{n+c},f,U)
                                 S ← S − U + {V_{n+c}}
                                 exitloop(i-loop)
                if S = OldS
                then return(null)
            return(e)
```

Function Verify_Function(V,X,f): **Boolean**
(* Given V, a set of variables, f a function on a subset of these variables, and X, another subset of variables in V that are believed to be irrelevant, this function returns **true** if f is verified to be a constant function on data tuples when each of the variables in X is varied; otherwise, **false** is returned. *)

for all $x \in X$
  $\{t_1,t_2\} \leftarrow$ Get_Data(V,X–$\{x\}$,$\{x\}$,2)
  If $f(t_1) \neq f(t_2)$
  then return(false)
return(true)

Function Get_Data(V,B,A,m): Tuple_Set
(* This function returns a set of m data tuples containing values for the variables in V, with variables in set B constrained to their basic values and variables in set A constrained to alternate values from their basic values. *)

$C \leftarrow \bigcup_{b \in B} \{(b,\text{Basic_Value}(b))\} \cup \bigcup_{a \in A} \{(a,\text{Alternate_Value}(a))\}$
$D \leftarrow$ Get_Tuples(V,C,m)
return(D)

Algorithm Reduction1 is organized as follows. Each iteration of the first loop (the S-loop) either reduces the number of variables in S remaining to be related together or the algorithm fails. If any variables are successfully related together by the inner loops, S is updated by removing those variables and adding one new variable, the inner loops are immediately interrupted, and another iteration of the first loop begins. Otherwise, the algorithm terminates and returns null.

The second loop (the i-loop) controls the size of the reduction made to S. During an iteration of this loop, an attempt is made to relate i variables together. Each iteration of the third loop (the U-loop) tries to combine a specific set U of i variables. A set of data tuples is provided by Function Get_Data such that the data tuples have a variety of values for the i variables in U and constant values for the other variables. A match is then attempted between these data tuples and several parameterized functional forms (or prototypes) on i variables. Each iteration of the fourth loop (the p-loop) uses Function Match_Prototype to try to match one prototype with the data tuples and thereby identify a function f on the variables of U that has a constant value for the data tuples. If this occurs, an equation e is created between the function and the constant value. Both f and e are returned by Match_Prototype. Then the function f is further tested (by Function Verify_Function, which is presented separately) on other data tuples. If f matches these data tuples as well, a synthetic variable V_{n+c} is created and defined to have the value of f applied

to the variables in U. Also, the variables in U are removed from S, the synthetic variable V_{n+c} is added to S, and the inner loops are terminated.

Algorithm Reduction1 includes calls to several functions:

1. *Function Get_Data* obtains suitable data tuples by constraining some variables to their basic values and other variables to alternate values. Alternate values are not used when obtaining the original set of data tuples, although they are used when verifying a function. Functions Basic_Value and Alternate_Value respectively return the basic value of a variable and an alternate value for a variable. These functions are used to encapsulate Wu's vague assumption about basic values, which is described in more detail in Section 3. Function Get_Tuples returns a specified number of data tuples containing values for a set of variables such that the values meet a specified set of variable-value constraints.

2. *Function Match_Prototype* applies an appropriate method to try to match a prototype p to all data tuples in D. In particular, Match_Prototype tries to find values for the parameters of p such that the resulting function f has a constant value on all data tuples in D. If successful, an equation e between f and the constant is created, and both f and e are returned; otherwise, **null** is returned for both. For example, if p were $f(x,y)=x^{r_1}+c_1y^{r_2}$, Newton's method could be used. If successful, the method would give values for the parameters r_1, c_1, and r_2, as well as the constant value of the function on the data tuples. The choice of an appropriate method is discussed in more detail in Section 3.

3. *Function Verify_Function* detects cases where the suggested function is not a constant function, by examining its values for other pairs of data tuples. Verify_Function performs the verification step of the Reduction method and is discussed in more detail in Section 3.

4. *Function Synthesize_Variable* defines a new synthetic variable (i.e., adds a new element to V) whose values are related to the values of the variables in U by function f. An implementation of Synthesize_Variable could immediately calculate values for V_{n+c} for all data tuples if they were known, but it would more likely create a procedure for automatically calculating values for V_{n+c} as data tuples are accessed. At most $n-1$ synthetic variables will ever be defined because S starts with n variables and each time one variable is defined and added to S, at least two variables are removed.

2.2. Complexity Analysis

In this subsection, the computational complexity of each of Basic and Universal Reduction is considered. The worst-case time complexity (hereafter *complexity*) is used as a measure of the computational complexity.

The complexity of Universal Reduction, denoted C_U, is simply the complexity of Algorithm Reduction1, and the complexity of Basic Reduction, denoted C_B, is the complexity of Algorithm Reduction1 with i restricted to 2. The effect on the complexity of the assumption of binary-partition decomposability can easily be seen by comparing C_U and C_B.

To simplify the analysis, the complexity of Algorithm Reduction1 is determined by examining the complexity of the three outer loops (the S-loop, i-loop, and U-loop) separately from the complexity of each iteration of the U-loop. Let $C_{U,1}$ and $C_{B,1}$ be the complexities of the outer loops for Universal Reduction and Basic Reduction, respectively; let $C_{U,2}$ and $C_{B,2}$ be the corresponding cost for each iteration of the U-loop. Since $C_{U,1}$ specifies how many times the body of the U-loop is executed and $C_{U,2}$ specifies the cost of each such execution, $C_U = C_{U,1}C_{U,2}$; similarly, $C_B = C_{B,1}C_{B,2}$.

2.2.1. Complexity of the Outer Loops

One can easily see that the worst-case complexity of Algorithm Reduction1 is exponential by examining its second and third loops (the i- and U-loops), which together try all subsets of set S except those that contain 0 or 1 elements. Thus, the body of the i-loop is executed $2^{|S|}-|S|-1 = O(2^{|S|})$ times. The original and maximum value for $|S|$ is n, the original size of V. If the i-loop is repeated until $i = |S|$ without altering S, then $|S|$ remains equal to n. In this case, the body of the i-loop is executed $O(2^n)$ times. The formulation of the algorithm assumes that some prototypes exist in all P_i for i up to n; otherwise some iterations of the i-loop will be wasted. Notice that in this worst case, the outermost loop (the S-loop) is executed only once. Thus, $C_{1,U} = O(2^n)$ for Universal Reduction.

Basic Reduction corresponds to Algorithm Reduction1 with the i-loop removed and the U-loop restricted to sets such that $|U| = 2$. In the modified algorithm, each change to S removes 2 variables and adds 1, and thus, the value of $|S|$ is $n, n-1, \ldots, 2$ on successive iterations of the S-loop. Since each iteration of the S-loop causes up to $\binom{|S|}{2}$ iterations of the U-loop,

$$C_{B,1} = \sum_{|S|=2}^{n} \binom{|S|}{2} = O(n^3)$$

2.2.2. Cost of One Iteration of the U-Loop

The cost of each iteration of the U-loop depends on the product of the number of prototypes in P_i and the cost of the methods used for matching and testing data tuples against a prototype. For Universal Reduction, let $C_{U,M}$ and $C_{U,T}$ be the costs of matching and testing, respectively; similarly, for Basic Reduction let $C_{B,M}$ and $C_{B,T}$ be the corresponding costs. If the number of prototypes in each P_i is assumed to be constant (an assumption which is discussed in more

detail in Section 3.2.3), $C_{U,2} = C_{U,M} + C_{U,T}$ and $C_{B,2} = C_{B,M} + C_{B,T}$. Now let us look in more detail at the cost of matching and the cost of testing.

Cost of Matching: For Universal Reduction, the cost of matching depends on the complexity of Match_Prototype, which is used to find a function f by matching a prototype to the data tuples in D. The most complicated method considered by Wu is numerical analysis using Newton's method, which has quadratic convergence, if a satisfactory initial approximation is provided ([Burden *et al* 78], p.445). However, the method uses $O(k^3)$ operations at each step of the convergence, where k is the number of parameters in the prototype equation. If the number of parameters is at least the number of variables in the prototype, then $O(i^3)$ operations are necessary when i variables are present in U. In the worst case, $i = n$. Thus, the complexity is $O(n^3)$ for a prototype requiring Newton's method. If all or any minimum fraction of the prototypes require this method, then the complexity of evaluating a prototype is $O(n^3)$. If i is restricted to the value 2, the cost is $O(1)$ because in this case, k is independent of n. Thus, $C_{U,M} = O(n^3)$ and $C_{B,M} = O(1)$.

Cost of Testing: The complexity of the method used for testing is given by the complexity of Function Verify_Function. For the sake of analysis, assume that an equal number of prototypes are present in each P_i for $i = 1,...,n$; therefore, the number of prototypes is a constant and can be ignored. The loop in Verify_Function is executed $|S - U|$ times, but the sizes of S and U vary in a way that depends on the main algorithm. In the worst case for Universal Reduction, $|S|$ remains fixed at n, but $|U| = i$ ranges from 2 to $|S|$. The loop in Verify_Function is executed $n - i$ times for each value of i, for an average cost of $\sim\frac{n}{2} = O(n)$ in the context of Algorithm Reduction1's worst case. For Basic Reduction, the size of S varies from n to 2 and the size of U remains fixed at 2. The loop in Verify_Function is executed $|S| - 2$ times for each value of $|S|$, for an average cost of $\sim\frac{n}{2} = O(n)$ in the context of Basic Reduction. For both Universal and Basic Reduction, each iteration of the loop in Verify_Function requires $O(n)$ operations because tuples of n values are fetched and then function f is computed on these tuples of n values.[5]

---

[5]Since some values are held constant, some of the n values do not need to be examined in every tuple, but in the worst case, U includes during each iteration a synthetic variable defined in terms of all variables removed from S. Values from all these variables must be examined in each data tuple, and the number of such variables rises incrementally from 2 to n. Thus, $\sim\frac{n}{2} = O(n)$ values are examined in each data tuple.

Therefore, $C_{U,T} = C_{B,T} = O(n^2)$.

Summary for *U*-Loop For Universal Reduction, the cost of one iteration of the *U*-loop is:

$$C_{U,2} = C_{U,M} + C_{U,T} = O(n^3 + n^2) = O(n^3).$$

For Basic Reduction, the corresponding cost is:

$$C_{B,2} = C_{B,M} + C_{B,T} = O(1+n^2) = O(n^2).$$

2.2.3. Summary of Complexity Analysis

The overall cost of Universal Reduction is:

$$C_U = C_{U,1}C_{U,2} = O(2^n n^3) = O(n^3 2^n),$$

and the overall cost for Basic Reduction is:

$$C_B = C_{B,1}C_{B,2} = O(n^3 n^2) = O(n^5).$$

Wu claimed a complexity of $O(n^4)$ for Basic Reduction, but he overlooked the cost of obtaining data tuples of n values and computing f for them. Thus, the assumption that i is always 2 which is made for Basic Reduction is not made simply "for the sake of clarity of description": it allows Wu to achieve polynomial time efficiency. Universal Reduction, the type of Reduction originally defined by Wu and used in his completeness theorem (Theorem 2.1 on p.376), has exponential complexity. The assumption that the target function is binary-partition decomposable allows i to be restricted to 2 and polynomial complexity to be obtained for Basic Reduction. Similarly, by limiting other features of Algorithm Reduction1, such as the total number of prototypes, other polynomial bounds could be obtained.

3. Analysis of Assumptions

In the design of the Reduction method, Wu made several unstated assumptions. In this section, these assumptions are explicitly documented and their usefulness and plausibility are discussed. Since an induction problem can be defined as determining whether or not a set of methods are sufficient to choose a hypothesis which describes a set of examples, Wu's assumptions will be discussed with respect to the *examples*, the *hypotheses*, and the *methods*.

3.1. Examples

Recall that in the numeric-data analysis problem, the examples are data tuples giving numeric values for a set of variables. Four relevant considerations are the *type* (positive or negative), the *format*, the *presentation*, and the *number* of the examples.

3.1.1. Type of Examples

Wu assumes that all examples are positive examples[6]. In other words, any tuple of data values that is available for access is a valid instance of the target equation. No mention is made of negative examples, but they deserve consideration because Wu's assumption about the order of the examples requires some treatment of negative examples, as is explained more fully in the following discussion.

3.1.2. Format of Examples

Wu implicitly assumes that each example can be described by a tuple of attribute-value pairs and that exactly the relevant set of attributes (variables) is specified. Wu states that the values are "numeric", which likely means that they are selected from a continuous interval (i.e., "interval values" in the standard nomenclature of data analysis). The units (or dimensions) of the values are not mentioned or used in any way; thus, the units are implicitly assumed to be unknown, irrelevant or nonexistent. This assumption is standard in data analysis, but in [Kokar 88], the search for relevant attributes is motivated by a dimensional analysis using the units of the target function (say $kg^1m^{-1}s^{-2}$) and the units of the known attributes. In the Reduction method, similar information could be used to constrain the search for a function by considering only the function forms involving compatible combinations of the variables' units.

---

[6]A positive example is a valid instance of the hypothesis being sought; this usage should not be confused with the use of "positive" in "positive integer".

The data tuples may be incomplete, according to Wu, and may contain incorrect or imprecise values. Unfortunately, he does not describe any way of coping with these types of data tuples and values. Lacking this information, the data tuples will instead be assumed to complete with correct and precise values.

3.1.3. Presentation of Examples

In general, the order in which examples are presented to an inducer can be either *controlled* or *uncontrolled*. With *controlled example presentation*, the inducer has some degree of control over the order in which examples are presented. One method of exerting control is to place constraints on the next data tuple to be presented. The constraints might take the form of restricting the values acceptable for a variable. A variable might be restricted to be one specified value or one of a specified range or set of values. Similarly, it might be restricted not to be a specified value or values. With *completely controlled example presentation*, the inducer completely constrains the next tuple by specifying all values in the tuple and then is informed of whether or not the tuple is valid. With *subset controlled example presentation* (hereafter *subset presentation*), values are specified for any non-empty, proper subset of the variables and the next tuple is a complete tuple including all these values. With subset presentation, the specified subset of values may be inconsistent, i.e., a positive example of the function including these values may be impossible. In this case, a special "no available tuple" value might be returned.

When subset presentation is used in a practical setting, several other factors must be considered. The set of examples actually available may happen to contain no tuples with the specified subset of values. Again, a special "no available tuple" value might be returned. To avoid such situations, a restricted form of subset presentation may be used in which some of the variables are identified by the user as available for control by the inducer. In this case, values must always be specified for only a particular subset of the variables. These variables are called the *predictor* or *independent* variables and the other variables are called the *response* or *dependent* variables. In this case, control over the examples consists of specifying the values for the predictor variables and then a tuple is presented with these values plus values for the response variables.

Wu has little to say about the presentation of examples, but implicitly makes three assumptions: (unrestricted) subset presentation is possible, negative examples are never

presented, and duplicates among tuples are not significant. The assumption of subset presentation greatly restricts the applicability of the Reduction method because data analysis often must be applied to a finite, predetermined set of data tuples. Also, in cases where new data tuples may be obtained on demand, such as scientific experiments, only a particular subset of the variables may be amenable to control. For example, in an experiment involving evaporation of a fluid, one can control an independent variable, such as the volume of the fluid, but one cannot directly control a dependent variable, such as the time required for evaporation. Reduction cannot be applied in such cases.

Wu apparently does not allow for negative examples (i.e., information about tuples that are not instances of the target equation). His method of testing allows the inducer to arbitrarily constrain the values of next tuple before requesting a tuple, but he does not describe how to handle cases where no consistent data tuple is possible that meets the specified constraints.

Finally, Wu apparently assumes that duplicates among data tuples are not significant. But in many experiments, duplicate data tuples are commonly produced, and the number of occurrences of a tuple is significant. Since Reduction does not consider the number of occurrences of a tuple, a data conversion might be performed before Reduction is applied to prevent information from being lost. For example, the duplicate tuples might be avoided by converting the format of the tuples to include a variable to uniquely identify the tuples or to represent the number of occurrences.

3.1.4. Number of Examples

Assumptions about the number of examples can be discussed with respect to the size of the example space (the number of different data tuples which might be accessed for a given target equation) and with respect to the number of data tuples accessed in the worst or typical case.

As can be gathered from the above discussion, Wu implicitly assumes that the example space is infinite. Since subset presentation is used, the presenter must either have many tuples already available or the ability to create tuples as required, because a tuple must be presented which satisfies any constraint the inducer specifies. As already noted, this assumption does not correspond to many practical cases.

The number of data tuples accessed by Algorithm Reduction1 depends on the number of

variables being considered and on the number of data tuples required to *instantiate a prototype* (i.e., choose values for parameters in a function form to produce a function) and to verify the resulting function. The total number of data tuples accessed is a function of the number of variables n; this function is closely related to the complexity of the algorithm used.

Wu makes no mention of the number of data tuples required to instantiate a prototype. In his example, he uses 4 data tuples, but in general, different methods for instantiating a prototype may require different numbers of data tuples. The formulation of Algorithm Reduction1 assumes that a constant $\#D$ can be specified in advance to represent the maximum number of tuples needed for any method of instantiating a prototype. An alternate approach would be to include with each prototype a specification of the number of tuples required to instantiate it. As the first step of the procedure for instantiating a prototype, the number of tuples available could be checked and more obtained if necessary.

With the assumption that at most $\#D$ tuples are ever required in the match data, the total number of data tuples needed can now be specified. Each iteration of the U-loop in Algorithm Reduction1 requires $\#D$ data tuples and each iteration of the loop in Function Verify_Function requires 2 verification data tuples. Since, the loop in Function Verify_Function is executed at most n times and Function Verify_Function is called at most once for each iteration of the U-loop, each iteration of the U-loop requires at most $\#D + 2n$ data tuples. In the worst case, Universal Reduction, requires $C_{U,1}$ iterations of the U-loop (see Section 2.2), and thus the number of data tuples required is $(C_{U,1})\ (\#D + 2n) = O(2^n)O(n) = O(n\,2^n)$. Similarly, in the worst case, Basic Reduction requires $(C_{B,1})\ (\#D + 2n) = O(n^3)O(n) = O(n^4)$ data tuples.

Finally, it should be noted that in practice, the number of data tuples required to verify functions will depend on how quickly spurious functions are rejected by the verification method. If, as seems likely, only a few pairs of verification data tuples will be examined before a spurious function is detected, this cost be considered a cost and combined with $\#D$. Thus, in practice, only n^3 data tuples may be needed for Basic Reduction.

3.2. Hypotheses

Significant assumptions are made by Wu about the hypothesis space for the numeric-data analysis induction problem. Wu implicitly assumes that there exists a nontrivial function on the complete set of variables that yields a constant value. This assumption prevents the hypothesis from being several constraint equations on subsets of the variables. Briefly, the hypothesis space consists of target equations whose target functions are composed of functions derived from the prototypes. In the remainder of this subsection, a set of constraints on the hypothesis space is accumulated by considering each of the following in turn: the target functions, the relation the composition of target functions and the prototypes, the number of prototypes, and the choice of prototypes.

3.2.1. Target Functions

Wu states assumptions about target functions using his terms "divisible" and "primitive function," which he defines as follows (p.375):

> A function $f(X_1, X_2, \ldots, X_n)$ $(n > 2)$ is *divisible*, if
> $f(X_1, \ldots, X_n) = p(X_1, \ldots, X_m) \ @ \ q(X_{m+1}, \ldots, X_n)$ $(0<m<n)$},
>
> where @ is a binary operator.
>
> A *primitive function* is a non-divisible function.

These definitions do not correctly describe the limitations placed on the hypothesis space by the Reduction method. The notation used in Wu's definition of "divisible" seems to require that the variables are ordered and that a partition is made at some point in the ordering. However, this ordering requirement was probably not intended by Wu and is not consistent with his further explanation of the Reduction method; therefore, it is not used in the present paper. On the other hand, the partitioning of the set of variables is essential to Wu's method and is preserved. Unfortunately, the explanation of the partitioning in terms of a "binary operator" describes the partitioning only for Basic Reduction, not Universal Reduction. According to Wu's definition, the variables are partitioned into only two subsets, but Universal Reduction requires up to n subsets. Also, the vagueness of the term "operator" leads to an ambiguity which will be discussed shortly. For these reasons, a new term, "binary-partition decomposable", will be defined to replace "divisible".

A distinction between the hypothesis spaces for Universal Reduction and Basic Reduction is created by differing assumptions about the composition of the target functions in terms of

primitive functions. For Universal Reduction, Wu states that any target function can be decomposed into at least one primitive function, but this statement is not significant because, in the worst case, the primitive function is simply the target function itself. Similarly, his Theorem 2.1, which states that "discovering any formula is equivalent to discovering its primitive functions" is not significant because it simply says that, in the worst case, discovering a target function is equivalent to discovering the target function itself. This worst case corresponds to the case where no constant-valued function is found for any proper subset of the original variables that matches the data tuples, i.e., where i reaches $|S|$ while S is still equal to V in Algorithm Reduction1. Therefore, the hypothesis space for Universal Reduction is not restricted by assuming that the target function is a combination of primitive functions.

However, another of Wu's assumptions does significantly restrict the hypothesis space for Basic Reduction. After discussing primitive functions, Wu states that "[g]enerally, the most common laws' primitive functions are binary functions."[7] Then he chooses to focus on the cases where the primitive functions are binary, supposedly "for the sake of clarity of description". It is not clear what would qualify as a "common law", but certainly many functions do not have primitive functions that are binary. For example, $f(x,y,z)=xy+xz+yz$ cannot be decomposed into binary functions; i.e., there are no functions f and g such that $f(x,y,z) = g(v_i,h(v_j,v_k))$ for any assignment of x, y, and z to one each of v_i, v_j, and v_k. Thus, limiting primitive functions to binary functions restricts the hypothesis space for Basic Reduction to a particular class of functions.

To clarify the limits of the hypothesis space for Basic Reduction, let us define a *binary decomposable* function as follows:

> **Definition:, Continue** A function $f(V_1,...,V_n)$ is *binary decomposable* if $n \leq 2$ (such a function is called a *primitive binary function*), or if $n>2$ and there exist functions f', g, and h such that
>
> $f(V_1,...,V_n) = f'(g(V_a,...,V_b),h(V_c,...,V_d))$;
> $\{V_a,...,V_b\} \neq \emptyset$;
> $\{V_c,...,V_d\} \neq \emptyset$;
> $\{V_a,...,V_b\} \cup \{V_c,...,V_d\} = \{V_1,...,V_n\}$; and
> $g(V_a,...,V_b)$ and $h(V_c,...,V_d)$ are both binary decomposable.

---

[7] Wu does not define the term "binary function", but presumably he means a function on two variables.

(The function *f'* used in the definition is called the *composition function*.) A function is *binary-partition decomposable* if it is binary decomposable and $\{V_a,...,V_b\} \cap \{V_c,...,V_d\} = \emptyset$ at each step of the decomposition. In other words, a binary-partition decomposable function can be decomposed into functions involving at most two variables by partitioning the variables into two nonempty subsets at each step of the decomposition. Wu's assumptions restrict the hypothesis space to the class of target equations whose target functions are binary-partition decomposable according to the definition just given.

3.2.2. Prototypes and Target Function Decomposition

To describe the hypothesis space further, we need to consider the set of prototypes and their relation to the process of decomposing the target function, and in particular their relation to the primitive binary functions and the composition functions. For any domain, Wu assumes that "a limited number of prototypes can cover almost all [of the] common laws", and on the basis of this vague assumption defines a set of prototypes.

Wu does not make clear the relationship between the prototypes just mentioned and the "binary operators" referred to his definition of a divisible function. It seems likely that the "binary operators" are either the familiar binary operators, such as the addition and multiplication operators of arithmetic, or else they reflect functions produced by instantiating the prototypes. In the first case, the composition function is restricted to the familiar binary operators and thus the prototypes are relevant only to the primitive binary functions. This interpretation further restricts the hypothesis space by restricting the set of target functions to binary-partition decomposable functions whose composition functions correspond to applying a single, familiar binary operator. In this case, functions involving only original variables are drawn from a different set than functions involving any synthetic variables.

In the second case, where the "binary operators" referred to in the definition of a divisible function are assumed to be equivalent to the operation defined by a function derived from a prototype, the term "binary operator" is confusing and unnecessary: the functions should simply be defined as the set of functions produced by instantiating the parameters in the prototypes to constants. In this case, the prototypes are relevant to both the primitive binary functions and the composition functions, and the set of target functions is restricted to binary-partition decomposable functions where the primitive binary functions and composition functions are

restricted to instantiated versions of the prototypes. This second interpretation will be adopted in the remainder of this paper, unless otherwise noted, because it avoids restricting the hypothesis space unnecessarily and provides a uniform treatment of original and synthetic variables.

3.2.3. Number of Prototypes

In Algorithm Reduction1, P represents the complete set of prototypes and P_i represents the set of prototypes on i variables. If at least one prototype is included for each possible value of i, then the number of elements in P will be at least $O(n)$ because Algorithm Reduction1 tries subsets of up to n variables. Also, if each variable appearing in a prototype has at least one parameter associated with it, then each prototype for i variables will include at least i parameters. Thus, for a prototype for n variables, the number of parameters will be at least $O(n)$.

Wu apparently assumes that the number of prototypes in each P_i is a constant independent of n. The prototypes he gives for P_2 match functions which are sums of variables, but not sums of products of variables. For example, $x+y$ matches against one of his prototypes, but $x+xy+y$ does not. To allow matches with such functions, additional prototypes would have to be added, and the number of additional prototypes needed would increase exponentially with the number of variables because of the many combinations of variables that are possible. (Alternatively, a single prototype could be used with all combinations of variables, but the number of terms, and thus, normally also the number of parameters in the prototype, would be exponential in the number of variables.) Thus, the assumption that the number of prototypes is constant may not be a good one, especially if prototypes are created automatically.

3.2.4. Choice of Prototypes

Wu considers only P_2 because with the assumption that the function is binary-partition decomposable, only prototypes of 2 variables need to be considered. He explicitly suggests four prototypes for P_2:

1. $f(x, y) = x^{r_1} + c_1 y^{r_2}$
2. $f(x, y) = x^{r_1} y^{r_2}$
3. $f(x, y) = (x^{r_1} + c_1)^{r_2} (y^{r_3} + c_2)$
4. $f(x, y) = (x^{r_1} + c_1)^{r_2} + c_2 (y^{r_3} + c_3)^{r_4}$

The first of these prototypes is a special case of the fourth and the second is a special case of the third. Presumably, the first two prototypes would be tried earlier because they are simpler. (The

ordering of the prototypes is considered briefly in Section 3.3.2 and discussed in more detail in Section 4.) Wu also suggests that "triangular functions, exponential functions, logarithmic functions and their combinations" could be added to the set of prototypes, but does not mention how many or which combinations are to be included. These suggestions correspond to assuming that, for each domain, the hypothesis space can be further restricted by limiting the set of functions used for primitive binary or composition functions to a set of functions with only a small set of forms. This assumption is not unusual in data analysis; for example, in statistical inference, one often begins with the assumption that the target function is a linear combination of the variables and small powers of the variables.

3.2.5. Summary of Assumptions about the Hypothesis Space

In summary, the hypothesis space for Universal Reduction is restricted to target functions which can be decomposed into functions, each of which is an instance of one of a constant-sized set of prototypes P. Similarly, the hypothesis space for Basic Reduction is restricted to binary-partition decomposable target functions, such that each primitive binary or composition function used in the composition is an instance of a constant-sized set of prototypes P_2.

3.3. Method

Wu's method for relating a subset of the variables consists of four steps: obtaining the *match data*, a set of data tuples for matching with prototypes; finding a constant function consistent with the match data; testing the function on *verification data*, another set of data tuples; and updating the set of variables.

3.3.1. Obtaining Match Data

To obtain the match data, Wu uses the idea of "basic data" (hereafter called *basic values*), in combination with subset presentation. Wu assumes that a basic value is available for each variable. Since the idea of basic values is introduced in the midst of an example, the origin of the basic values is not clear from Wu's presentation. Two possible sources of the basic values are the inducer and the presenter; each of these possible sources are now considered in turn.

The basic values might be chosen by the inducer before any data tuple is presented. This approach to induction seems misguided or incorrect because the inducer has no information on which to make a choice, and thus must choose a set of basic values at random. It is probable

that the set of basic values is not consistent with the target equation (i.e., that if they were treated as a data tuple, they would not be an instance of the target equation) since the values were selected at random. Choosing an arbitrary subset of such a set of values may constrain presentation so that no tuple in the example set can be provided, i.e., force only negative examples to be provided. Wu does not consider this possibility in his description. If Wu's intention was that the basic values are to be chosen by the inducer, a better approach would have been to use the set of values from the first (positive) data tuple, which is guaranteed to be consistent with the target equation.

Alternatively, the basic values might be assumed to be provided by the presenter as additional information. In this case, the basic values are either particularly useful values or simply random choices. If they are particularly useful values for the variables, a major assumption is being hidden under the name "basic values". The alternative is that the basic values are chosen at random. If the set of basic values is a random choice of a set of values that is consistent with the target equation, providing these values is equivalent to providing an initial data tuple. If the set of values is not consistent with the target equation, they are no more useful than a random choice made by the inducer. Thus, either the basic values represent a major assumption about aid from the presenter or they are unnecessary.

In Algorithm Reduction1, Function Get_Data is used to obtain all data tuples. When relating a subset U of the variables in S, data tuples are obtained such that the variables in U are varied and the variables in $S-U$ are held constant. Adopting the view that basic values are unnecessary, the call to Get_Data to obtain the match data could be replaced by the call Get_Match_Data(V, $S-U$, #D) to the following function:

```
Function Get_Match_Data(V,Fixed,m): Tuple_Set
    d ← Get_Tuple(V,∅)
    C ← ∅
    for v ∈ Fixed
        C ← C ∪ Variable_Value_Pair(v,d)
    D ← {d} ∪ Get_Tuples(V,m − 1,C)
    return(D).
```

Get_Match_Data first obtains one data tuple and then uses it to constrain the values of other tuples. It is assumed that the constraints are stored in C for later use in testing. Get_Tuple gets one tuple for the set of variables given by its first argument that is consistent with the set of constraints specified by its second argument. Given a variable and a data tuple,

Variable_Value_Pair returns a pair consisting of the variable and its value in the data tuple. Get_Tuples gets a set of tuples containing the number of tuples specified by its second argument, given the constraints specified by its third argument.

One remaining issue that needs to be resolved is what it means to hold a synthetic variable to a constant value while obtaining data tuples. Two possibilities are as follows:

1. All original variables that are included directly or indirectly in the definition of the synthetic variable are held to constant values; or
2. The values of the original variables may vary, but when they are combined according to the definition of the synthetic variable, they yield a constant value.

Either of these interpretations may be adopted for Reduction, and implemented in the Get_Tuple and Get_Tuples functions, but clearly the first possibility yields a more manageable situation.

3.3.2. Finding a Function

To find a constant-valued function on a subset U of the variables that is consistent with the match data, a match is attempted between the match data and each of a set of prototypes. Two relevant issues are the technique used for matching and the order in which the prototypes are examined. The latter issue is relevant because the algorithm must choose some order to examine the prototypes, and the order chosen can significantly effect the quality of the match and the efficiency with which it is found. However, Wu does not discuss prototype ordering, and a discussion of this issue is postponed until Section 4, where a strategy for ordering prototypes is given. Now let us consider the techniques used for matching a prototype to the match data.

A match is obtained if the parameters of a prototype can be given constant values such that the resulting function yields a constant value for all the tuples. To find values for the parameters, three methods are proposed in [Wu 88]:

1. the user "provides an analytical expression for each ... parameter"
2. "enumerate the most possible [sic] values of some parameters"
3. "use a numeric solution such as Newton's Iteration".

Wu does not elaborate on these three methods, but each will be considered in turn here.

The term "analytical expression" has several meanings in mathematics, but the meaning that Wu most likely intended was "closed-form expression". If this meaning was intended, then the first method requires ways of calculating parameters for functions from data, and indeed closed-form expressions are known for some function forms. For a simple linear equation $y=ax+b$ and

two data tuples (x_1,y_1) and (x_2,y_2), the constants can be calculated as $a=(y_2-y_1)/(x_2-x_1)$ and $b=y_1-ax_1$. For Wu's second prototype, a closed-form expression might be derived as follows, using k as the constant value of the function:

$$x^{r_1}y^{r_2} = k$$

$$r_1 \ln x + r_2 \ln y = \ln k$$

$$\ln y = \frac{\ln k}{r_2} - \frac{r_1}{r_2} \ln x$$

This equation can be reformulated as a linear equation $y'=ax'+b$ where $y'=\ln y$, $x'=\ln x$, $a=\frac{-r_1}{r_2}$, and $b=\frac{\ln k}{r_2}$. Then from two data tuples, two values for each of x' and y' can be computed and used to solve the linear equation as described above. To obtain values for the parameters, set r_1 to some nonzero value, say 1, and solve for r_2 and k using the known values for a and b. The parameter r_1 can be set to an arbitrary nonzero value because when $r_1 \neq 0$, Wu's prototype is equivalent to $xy^{r_2} = k$, with an appropriate scaling of the constants r_2 and k. (The case of $r_1=0$ is not of interest because it results in a trivial function that does not depend on x.) The constants r_1 and r_2 can be found easily once a and b are known. If closed-form expressions are known for the parameters of the prototype, these expressions should be directly associated with the prototype when the prototype is included in set P. However, as Wu notes, "it is difficult to obtain an analytical solution in the case of complex prototypes" (p.377)[8]. In particular, if by analytical expressions he means closed-form equations, analytical expressions are not available for his other three prototypes. Therefore, the first method is useful in some cases but not generally applicable.

The second method, enumerating the most likely values, is best seen as a combination of a further restriction on the hypothesis space, wherein the set of possible values for the parameters is specified, and a method (an order) for trying them. Wu's proposal of the second method is inconsistent with his statement that "there are numerous binary functions and [it is] impossible to enumerate them one by one as ... is done in BACON and ABACUS. An alternate approach is to enumerate them one class ... [at a] time" (p.377). The second method does in fact enumerate the

[8]The word "complex" is taken to mean "complicated" not to suggest that both real and imaginary numbers may be involved.

binary functions one at a time rather than a class at a time. Also, it is likely to be slower than BACON and ABACUS because it is a generate-and-test method rather than a heuristic search.

The third method suggests using iterative numerical-analysis techniques to find values for the parameters. As with the first method, the particular iterative method should be associated with the prototype when it is chosen for inclusion. For such an approach, the values from the data tuples are used to create a set of equations (called "error functions") of the form $e(r_1,r_2,...,r_i) = 0$ that constrain the values of the parameters $r_1,r_2,...,r_i$. Then iterative techniques are applied to search for a set of values for the parameters that satisfies these equations, i.e., where the error functions are all zero. In practice, no such set of values exists because of machine arithmetic and imprecise or incorrect data. Instead, a set is found that minimizes the "total error", as measured by some metric, such as the sum of squares of the error. Using an iterative method requires the assumptions that the method will terminate if no values for the parameters can give a function that matches the data tuples, that appropriate starting values for the parameters can be selected, and that an appropriate termination condition can be specified.

Wu does not describe in detail how the methods of finding parameters should be chosen or combined. He states that the first type of method is the best, the second is the simplest, and the third is the most general. He also suggests that the third method should be used if the other two fail, which suggests an ordered sequence of methods. Based on this information, choosing from the three methods might be accomplished as follows: if analytical expressions exist for the parameters, the first method should be used; otherwise, try the second method for the specified values (if any); and if none of these produce a match, try the third method. A clearer way of organizing these methods would be to associate a procedure with each prototype that obtains values for its parameters. The procedures could range in complexity from simple calculations for linear equations to sequences of values to try and numerical analysis methods to apply for more complicated cases. In Algorithm Reduction1, the task of selecting a method for finding the parameters for a prototype is hidden in Function Match_Prototype. With the approach just outlined, Match_Prototype could simply call the procedure associated with the specified prototype.

3.3.3. Testing the Function

For testing that a function defined on the variables in U yields a constant value, Wu uses pairs of new tuples. Each pair is used to test that a different one of the variables not in U, but still in S, can be altered without affecting the match between the function and the data tuples. Let us refer to the variable being tested with a pair as the *candidate variable*. All variables in S except the candidate variable and all variables in U are held to the same values that they had in the match data. According to Wu's idea of basic values, the request for the first data tuple of the pair is therefore unconstrained for the variables in U, constrained to an *alternate value* (i.e., a value other than its "basic value") for the candidate variable, and constrained to basic values for the other variables. The request for the second data tuple of the pair is similarly constrained except the candidate variable is constrained to the same value as in the first data tuple of the pair. Wu does not identify the means of selecting the alternate values: as with basic values, the idea of having a value for a variable other than its basic value occurs only during an example. It is not clear whether there is only one alternate value for a variable, or if various values are used during the testing of different functions. As with basic values, the idea of alternate values is suspect: either it is a misguided way for the inducer to obtain another value for a variable, which might better be specified as a constraint that the variable not be equal to a certain value, or it is a subtle way of assuming a helpful presenter of examples. The former is assumed in the remainder of this paper.

The constraint that a variable not be equal to a certain could be added to the method of obtaining verification data tuples. To do so, the call to Get_Data in Function Verify_Function could be replaced by the call Get_Verification_Data(V,X,x,2) to the following function:

Function Get_Verification_Data(V,X,x,m): **Tuple_Set**
$C_2 \leftarrow C$
$c \leftarrow$ Delete_Constraint(C_2,x)
$d \leftarrow$ Get_Tuple(V,$C_2 \cup \sim c$)
$C_3 \leftarrow C_2 \cup$ Variable_Value_Pair(x,d)
$D \leftarrow \{d\} \cup$ Get_Tuples(V,$m-1$,C_3)
return(D).

This function obtains data tuples such that variable x has a different value from its values in the match data. C contains the constraints defined by Get_Match_Data. Given a set of constraints and a variable, Function Delete_Constraint returns the particular constraint on the variable after removing it from the set of constraints. The notation $\sim c$ denotes the reverse of constraint c.

Wu states as a theorem that if the function "still has a constancy [for the new pair] ... then in the normal case" the candidate variable does not appear in the function (p.378). The theorem is not useful because the function was defined by the inducer on two variables, neither of which was the candidate variable, and therefore, it cannot depend on the candidate variable. It seems probable that Wu meant to state his theorem as follows: If the function still has a constancy on the new pairs associated with all candidate variables, then "in the normal case" the function is a primitive function of the target function. This is the conclusion one would expect from the argument he gives before presenting the theorem. In fact, for the interpretation that the hypothesis space is binary-partition decomposable functions, the constancy indicates "in the normal case" that the function is a primitive binary function or a composition function in some decomposition of the target function. The meaning of the phrase "in the normal case" will be considered further after a discussion of some difficulties with Wu's approach to testing.

Wu's verification procedure is not sufficient to verify that the chosen function is a constant-valued function. Wu states that "a primitive function is a constant function" (p.377), but this is false, as will be shown shortly. He uses the following argument as the basis of his method (p.376):

> If $h(X_1,X_2)$ is a primitive function of $f(X_1,X_2,...,X_n)=c$, then, given a set of varying data between X_1 and X_2: $\{(d_{1,1},d_{2,1}),(d_{1,2},d_{2,2}),...,(d_{1,m},d_{2,m})\}$, obviously we have a system of equations:
> $h(d_{1,1},d_{2,1})=h(d_{1,2},d_{2,2})=...=h(d_{1,m},d_{2,m})$

Suppose that the target equation is $g(x,y,z,u,v)=(x+y)^2/z-uv=2$. According to Wu's Table 1 (p.375), the target function $(x+y)^2/z-uv$ has two primitive functions[9] $x+y$ and uv. Function g, when evaluated on data tuples <1, 2, 1, 1, 7>, <4, -1, 1, 1, 7>, and <-2, -1, 1, 1, 7> gives the constant value 2. However, the primitive function $x+y$ yields the values 3, 3, and -3. Thus, this primitive function is not a constant-valued function. Wu's equation is not true in general for functions.

Wu's method cannot reliably distinguish between $x+y$ and $(x+y)^2$ as primitive functions. For some sets of data tuples for which $(x+y)^2$ is a constant-valued function, $x+y$ will also be a constant-valued function, but for other sets it will not. Suppose that the actual constant-valued

---

[9]Following Wu's definition of primitive functions with the interpretation of "binary operator" as ordinary arithmetic operator would give $(x+y)^2$ and uv as the primitive functions. If these primitive functions are used, another error results, as will be shown shortly.

function on x and y is $(x+y)^2$ and that a match is attempted using $x+y$ as a prototype. In some cases, such as the example just given, $x+y$ will be rejected as not matching the match data. In other cases, $x+y$ will match the match data, but fail to match the verification data. For example, if the match data tuples are {<1, 2, 1, 1, 7>, <4, -1, 1, 1, 7>, <3, 0, 1, 1, 7>}, and the verification data tuples for candidate variable z are {<1, 2, 2, 1, 7>, <-2, -1, 2, 1, 7>}, $x+y$ has a constant value on the match data, but not on the verification data. In still other cases $x+y$ will match both match and verification data, as it would if the sum of x and y were 3 in all verification data tuples. Nonetheless, the actual constant-valued function is $(x+y)^2$. Thus, when a match is tried against a prototype $x+y$, it may be either accepted or rejected depending on the particular data tuples obtained, not on whether it is a primitive function.

If the term "binary operator" was interpreted to mean an ordinary arithmetic operator (as discussed in Section 3.2.2), then the primitive functions of $(x+y)^2/z-uv$ would be $(x+y)^2$ and uv. Wu's method may still fail in this case if the prototype $x+y$ is considered before the prototype $(x+y)^2$. Depending on the data tuples examined, $x+y$ might be accepted before $(x+y)^2$ is even considered.

In general, if the inverse of a mathematical function used in the actual constant-valued function is not unique (i.e., not itself a function), the verification procedure may reject the correct function. In the extended example just considered, the actual constant-valued function was $(x+y)^2$, and this function includes the squaring function, which has two inverses: the positive and negative square roots. Other functions, such as *sine*, have an infinite number of inverses. Thus, Wu's verification procedure is not sufficient to verify functions in general.

These examples suggest a possible meaning for Wu's "normal case" where the verification method works. In each example just given where the method failed, values were chosen that did not allow similar functions to be distinguished. The "normal case" may therefore be the case where these values do not arise. However, cases other than the "normal case" may arise with significant frequency if few data tuples are used when selecting a constant-valued function (i.e., if $\#D$ is small). For example, when searching for a function on x,y,z with $\#D = 3$, only 5 data tuples would be examined, and all the possible confusions between $x+y$ and $(x+y)^2$ could arise.

Thus, Wu's testing method has the advantages of simplicity and efficiency but the

disadvantage of possible incorrectness. Redefining the induction problem to avoid the possibility of incorrectness can be done. One could restrict the set of functions to which the method is applicable to those functions which have a single inverse for all values, but this would exclude many interesting functions, including those involving multiplication and division operations. Alternatively, one could assume that the data tuples presented are in "general position", i.e., that they will not appear to be instances of the wrong function. Also, to reduce the chance of error, the function could be checked with a larger set of data tuples (i.e., a larger $\#D$) before applying the verification method. A combination of extra checking with the assumption of general positioning of data is adopted in the revised Reduction algorithm presented in Section 4.

3.3.4. Updating Variables

The fourth step of Wu's method updates the set of variables S. He assumes that all variables in U can be removed once a constant-valued function on these variables has been discovered. If we grant that a target function on all the variables exists, that this function is binary-partition decomposable, and that the function finding and verifying methods correctly identify a suitable binary function (which they do not do in all cases, as just discussed in Section 3.3.3), then this method is correct and efficient. In other induction programs, variables are not removed because the same assumptions are not made. For example, in BACON, synthetic variables (called terms) are added but existing variables are left in place, which leads to a variable set that expands without bound. In ABACUS, existing variables are left in place when synthetic variables are added, but dependencies among the variables are recorded and no effort is made to keep one variable constant if one of its dependent variables is being varied. In both of these cases, the function is not assumed to be binary-partition decomposable; the assumption that the variables can be partitioned allows Wu to use an easy method of updating the set of variables.

4. Improved Algorithm for Basic Reduction

In this section, an improved algorithm for the problem addressed by Reduction is presented. First, a list is presented of the assumptions that must be made for Reduction to be applicable. Since the complexity analysis showed that Reduction is not efficient unless restricted, the assumptions also restrict the class of target functions to binary-partition decomposable ones. All these assumptions are required for Basic Reduction to be applicable, and all but the assumption of binary-partition decomposable functions are required for Universal Reduction to be applicable. Then Algorithm Reduction2, which provides a more efficient solution to the problem addressed by Basic Reduction, is presented. Following the algorithm, the worst-case time complexity for Reduction2 is shown to be O (n^4), in comparison to Basic Reduction's O (n^5). Finally, a method for ordering prototypes is suggeted.

The following assumptions are required for Basic Reduction and for Algorithm Reduction2:

1. the target equation can be written in the form

   $f(V_1,...,V_n) = c$

   where

   c is a numeric constant;

   function f is a nontrivial function on the variables $V_1,...V_n$;

   function f is binary-partition decomposable;
2. there exists a decomposition of function f such that each function used in the decomposition is an instance (with parameters given constant values) of a prototype in a set of prototypes P;
3. data tuples are complete, correct, and precise;
4. subset presentation is available for the data tuples;
5. duplicate data tuples are not significant;
6. a maximum number of data tuples $\#D$ needed to match any prototype is known for the prototypes in P.
7. the data tuples are in general position, i.e., they are not particularly chosen to disguise the target function;
8. each method associated with a prototype $p \in P$ is an algorithm capable of instantiating the parameters in p to produce a function f using $\#D$ data tuples if such a match is possible.

A major improvement could be made in the efficiency of the Reduction method by searching for a function between a set of variables only once. In Algorithm Reduction1, each iteration of the S-loop searches for a set of related variables by considering all subsets of S. However, after a few iterations, many subsets will have already have been considered and there is no need to

attempt to match them to a function again. This wasted effort is also present in Basic Reduction, where the same pair may be examined over and over to see if the two elements can be related together. To avoid this wasted effort, each pair of variables, whether original or synthetic, should only be considered once. Also, pairs of original variables should be considered first, and then synthetic variables should be considered in order of creation. From n original variables, at most $n-1$ synthetic variables can be created; let us numbered them, in order of creation, as $V_{n+1},\ldots,V_{2n-1}$. By considering each of the variables, in order of subscript, and trying to relate it to variables with lower subscripts, all pairs are considered only once and in an order that tries all combinations of previously existing variables before considering a newly created variable. If all pairs of variables with lower subscripts have been examined, but no new synthetic variable has been created, the target function is not binary-partition decomposable.

Algorithm Reduction2, presented below, is based on the method just described. This algorithm is more efficient than Basic Reduction because the a pair of variables is only examined once, not repeatedly. Functions Verify_Function, Get_Match_Data, and Get_Verification_Data, which have been presented and discussed earlier are included here for completeness.

Algorithm Reduction2

```
(* Given a set V of n variables {V_1,...,V_n} and access to data tuples providing values for
these variables, this algorithm either finds and returns a target equation or returns null if
no equation can be found. A set P of prototypes on two variables are also assumed to be
available. The number of data tuples to be used for matching against prototypes is given
by #D. *)

        c ← 0
        S ← V
        for i = 2,...,2n-1
j-loop:     for j = 1,...,i-1
                if V_j ∈ S
                then U ← {V_i, V_j}
                     D ← Get_Match_Data(V,S-U,#D)
                     for all p ∈ P
                         {f,e} ← Match_Prototype(p,U,D)
                         if f ≠ null and Verify_Function(V,S-U,f)
                         then c ← c+1
                              Synthesize_Variable(V_n+c,f,U)
                              S ← S-U+{V_n+c}
                              exitloop(j-loop)
        if |S| = 1
        then return(e)
        else return(null)
```

```
Function Verify_Function(V,X,f): Boolean
        for all x∈X
            {t1,t2} ← Get_Verification_Data(V,X,x,2)
            if f(t1) ≠ f(t2)
            then return(false)
        return(true)

Function Get_Match_Data(V,Fixed,m): Tuple_Set
    d ← Get_Tuple(V,∅)
    C ← ∅
    for all v∈Fixed
        C ← C ∪ Variable_Value_Pair(v,d)
    D ← {d} ∪ Get_Tuples(V,m−1,C))
    return(D)

Function Get_Verification_Data(V,X,x,m): Tuple_Set
    C2 ← C
    c ← Delete_Constraint(C2,x)
    d ← Get_Tuple(V,C2 ∪ ~c)
    C3 ← C2 ∪ Variable_Value_Pair(x,d)
    D ← {d} ∪ Get_Tuples(V,m−1,C3)
    return(D)
```

In brief, the the worst-case time complexity of Algorithm Reduction2 is:

$$\sum_{i=2}^{2n-1}\sum_{j=1}^{i-1}[1+(|S|-2)n] \le \sum_{i=2}^{2n-1}\sum_{j=1}^{i-1}[1+n^2] = O(n^4)$$

where $|S| \le n$ because S originally contains n elements and two elements are removed each one is added. This worst case corresponds to no satisfactory relation between any two variables being found until the combination of the last two variables is examined and a synthetic variable created, and then no combination being found for this synthetic variable until the last remaining original variable is examined, and so on.

The choice and order of the prototypes is a key factor in determining the usefulness of Reduction. As mentioned in Section 3.2, the four prototypes given by Wu suggest that ordering for efficiency is a relevant concern because the first two are special cases of the last two. Ordering prototypes will be considered in conjunction with choosing prototypes because a method is proposed for choosing prototypes that also yields suggestions for ordering these

prototypes.

It is difficult to specify a suitable set of prototypes. Any set of prototypes represents a trade-off between completeness and efficiency: the larger the set of prototypes, the more chance that it includes the right one for the data, but the longer it takes to search. A reasonable compromise would be a parameter which specified the level of thoroughness of search; then the number of prototypes could be selected to reflect the user's patience. Regardless of exactly how many prototypes are included, it is likely that many more than four prototypes would be needed. Of the prototypes given by Wu, only one (i.e., #2) has an analytical solution. To avoid invoking the relatively expensive process of iterative numerical-analysis techniques, additional prototypes with analytical solutions should be incorporated. For example, the simple linear equation $y=ax+b$ should be included. An initial set of prototypes could be chosen by giving common classes of equations, such as the common models used in statistical inference. As well, it would be useful to allow the user to specify a set of prototypes that must be included and a set that could be excluded, which might allow the selection of a more useful set of prototypes for a particular application.

The ordering of the prototypes is important for Reduction because a linear search of the prototypes is made to find a match with the data tuples, and this search is interrupted if a successful match is found. However, Wu does not describe any method for ordering the prototypes. The number of elements in the set of prototypes is even more important than the order in which the prototypes are examined, because many searches are made through the whole set of prototypes without finding any match. Thus, a method is suggested here that involves both the selection and ordering of prototypes to produce a sequence L of prototypes. A prototype p_s that is a special case of a more general prototype p_g should only be included in L if a more efficient method is available for matching against p_s than for p_g, where the efficiencies could be determined by classes of time complexity, such as $O(n^2)$ versus $O(n^3)$, or by timing the matching procedures while Reduction is being used. If both prototypes appear in L, p_s should appear before p_g. If a number of special cases of p_g are included in this way, then the total cost of the special cases should be compared to the cost of the p_g. If the cost of the p_g is less than half of that of all the special cases combined, then only p_g should be included in L.

5. Conclusions and Suggestions for Further Research

In this paper, the Reduction method presented in [Wu 88] has been critically appraised. Wu's presentation of the method was vague or incorrect in places. The approach taken in the present paper was constructive: an attempt was made to identify the assumptions needed for the Reduction method to perform correctly and efficiently. As a result of this study, a number of conclusions can be drawn. Reduction is not an efficient, general-purpose method for discovering regularities in data. Reduction requires the existence of a function that depends on all of the variables and yields a constant value for all data tuples, but in actual situations, several separate constraint equations may be present. Unless some assumptions are made, the time complexity of Reduction is exponential, not polynomial, because not all target functions are binary-partition decomposable. Also, for such functions, Algorithm Reduction2, which was presented in Section 4 is preferable to Wu's Basic Reduction method because it is more efficient; Reduction2 is O (n^4) while Basic Reduction is O (n^5). Also, the testing method suggested by Wu may give incorrect results, as demonstrated by counter-example.

As well, the applicability of the Reduction method is seriously limited by the assumption that the data can be presented in an order determined by the inducer. Reduction requires subset presentation, i.e., that a suitable data tuple can be obtained simply by specifying constraints on a subset of its variables. In actual situations, subset presentation is unavailable because only a subset of the variables are amenable to control or more usually because induction must be performed on a given set of data presented in particular order.

The scope of the Reduction method is a broad range of data induction problems. Typically, statistical analysis attempts to fit one equation model to a set of data. For example, multiple linear regression looks for equations of the form

$$\Sigma_{i=0}^{n} a_i x_i$$

where the x_i's are variables and the a_i's are associated constants. Numerical-analysis techniques can be applied to select one of a family of equations by approximating the parameters in an equation form. The aim of the Reduction method is broader still: to co-ordinate the search among many equation forms.

For this task, Wu recognized the need to incorporate numerical-analysis techniques to match

data tuples to possible models for the target equation, but did not incorporate statistical-inference techniques. Statistical inference includes various models of equations that resemble those present in Wu's suggested set of prototypes for Basic Reduction. For example, the model for multiple linear regression that was described above gives binary-partition decomposable functions. Variations of this form with extra variables such as x_i^2 or x_i^3 are also tried when statistical inference is attempted in practice. Again, the relevant functions are binary-partition decomposable. Thus, Algorithm Reduction2 handles a larger class of functions than multiple linear regression, but is much less efficient because the variables are considered two at a time rather than all once, as is done in statistical inference using matrix methods. A way of incorporating statistical inference techniques efficiently into Algorithm Reduction2 is needed.

Much work remains on the numeric-data analysis induction problem. Such a problem might include capturing regularities that are specified by sets of inequalities or sets of equations, rather than simply by one equation. For example, if it is discovered that two variables increase together, but no function exists between their rates of increase, constraint inequalities on their derivatives could be produced. The ability to discover such constraint inequalities would make the method more appropriate to data-analysis problems in general. Even if the description of the data were assumed to be in the form of several equations on the variables, rather than a single target equation, a broader range of problems would be covered.

With regard to this last type of problem, it should be noted that Algorithm Reduction2 could be easily modified to handle a simple version of this case. When several equations are needed, each equation is on a subset of the variables in V. Two cases arise depending on whether or not the subsets are disjoint.

For the case where the subsets are disjoint, a slightly modified version of Algorithm Reduction2 can solve the problem (assuming that the conditions listed in Section 4 are met). The algorithm will proceed as normal but stop with more than one variable remaining in S and return null. With a slight change to the algorithm to return any original variables still present in S along with equations corresponding to the synthetic variables in S rather than null, Algorithm Reduction2 will solve the problem. Some original variables are returned along with equations, because none of these variables have been related to any other variable. It is worth noting that ABACUS, which is described in [Falkenhainer and Michalski 86] as being able to handle problems where several

equations are needed, only handles the case where the equations are defined on disjoint subsets of the variables.

For the case where the subsets are not disjoint, the modified version of Algorithm Reduction2 could easily fail to find all subsets because each variable is eliminated as soon as the first equation is found that involves this variable. Afterwards, no other equation involving the variable can be found because the variable is no longer even being considered. One means of overcoming this problem would be to apply backtracking to find equations for all possible subsets. Backtracking might be incorporated by simply restarting the algorithm, whenever it would otherwise terminate, with all variables back in S and then rejecting any combination that includes a subset that has already been related together. To make the search more efficient, dependency directed backtracking could be used: each time the search reachs a dead end, any new equations that had been discovered would be recorded before backtracking to the relevant choice point. Unfortunately, adding backtracking would change the worst-case time complexity to exponential from polynomial.

Universal Reduction is applicable to complicated target functions, but Basic Reduction and Reduction2 are applicable only to binary-partition decomposable ones. Exploring the usefulness of permitting limited complexity, such as combinations of three variables, could also be done. The method used in Algorithm Reduction2 should be generalized to handle such cases by adding more loops.

Reduction's testing method has been shown to be incorrect; a new testing method is needed. The effectiveness of the current method could be measured, and if it is relatively effective, especially when supplemented with additional testing on more data tuples as suggested in Section 3.3.3, it could continue to be used, with backtracking to make changes to the function and retest when inconsistencies are found.

Overall, Wu's Reduction method is unsatisfactory from an Artificial Intelligence (AI) perspective because a large search space of variable combinations and prototypes is defined but no AI techniques are applied to increase the likelihood of a quick solution. In particular, no heuristics have been suggested for choosing the order in which variable combinations or prototypes are considered. Also, no way of incorporating knowledge about the appropriateness of the

prototypes for particular problems or domains was suggested.

When trying to identify a subset of the variables that can be related together by a function, a heuristic approach might be used. The following heuristic is also suggested for searching for binary-partition decomposable functions: each time that an attempt to find independence between two variables V_a and V_b is thwarted by a dependence between them and a third variable V_c, a dependency counter for the two pairs (V_a, V_c) and (V_b, V_c) should be incremented. Then, when choosing a pair of variables to search for a relation between, the pair with the highest dependence counter should be tried first. This heuristic uses previously discovered information about dependencies to guide the search for exact descriptions of the dependence.

An efficient order for searching the prototypes would improve Algorithm Reduction2. As suggested in Section 4, an ordering of prototypes according to increasing computational cost would help. If the resulting ordering was similar to the ordering of the prototypes from most probable to least probable, the Algorithm Reduction2 should be reorganized to consider the prototypes in this order, and for each prototype to search for a function on all possible combinations of two variables, before considering the next prototype. Unless some information is known about which variables are likely to be related, this organization of the algorithm would be more efficient because the expensive computations of the complicated prototypes could be avoided for the variable pairs before the correct pair, in return for doing inexpensive computations of the simpler prototypes for the pairs of variables after the relevant pair. As well, for each variable pair, the furthest prototype attempted could be recorded. As synthetic variables are created, the earlier prototypes could be tried on combinations of them and older variables, without redoing previous work.

An even better method of organizing the prototypes would incorporate information about how often the prototype actually matches data in general or in a particular domain. Such information could only be obtained by analyzing the relative frequency with which the prototypes match real sets of data. Also, the set of prototypes might be structured in a more complicated way than a simple linear ordering, to allow classes of prototypes to be included and excluded easily and to allow results obtained when using one prototype to guide the choice for the next prototype to be considered. Finally, some measure of the worth of functions derived from different prototypes

might be incorporated. For example, prototypes with fewer parameters are valued more highly by scientists.

Acknowledgements

I owe a special debt of gratitude to Michael Dyck for hours of discussion and several of the best ideas in this paper. This paper also benefited from discussions with Romas Aleliunas, Nick Cercone, Jim Delgrande, and Wilf Rosenbaum.

References

[Burden *et al* 78] Burden, R.L., Faires, J.D., and Reynolds, A.C.
Numerical Analysis.
Prindle, Weber, and Schmidt, Boston, MA, 1978.

[Falkenhainer and Michalski 86]
Falkenhainer, B.C. and Michalski, R.S.
Integrating Quantitative and Qualitative Discovery: The ABACUS System.
Machine Learning 1(4):367-401, 1986.

[Kokar 86] Kokar, M.
Determining Arguments of Invariant Functional Description.
Machine Learning 1:403-422, 1986.

[Kokar 88] Kokar, M.
Syntactic Equivalence in Concept Discovery.
In *Proc. of the First International Workshop in Change of Representation and Inductive Bias*, pages 268-278. 1988.

[Langley 78] Langley, P.
BACON.1: A General Discovery System.
In *Proc. CSCSI-78*, pages 173-180. Toronto, Ont., 1978.

[Langley *et al* 83] Langley, P., Bradshaw, G.L., and Simon, H.A.
Rediscovering Chemistry With the BACON System.
In Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (editors), *Machine Learning: An Artificial Intelligence Approach*, pages 307-329. Tioga Publishing Company, Palo Alto, CA, 1983.

[Langley *et al* 86] Langley, P., Zytkow, J.M, Simon, H.A., and Bradshaw, G.L.
The Search for Regularity: Four Aspects of Scientific Discovery.
In Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (editors), *Machine Learning: An Artificial Intelligence Approach, Volume 2*, pages 425-469. Morgan Kaufmann, Los Altos, CA, 1986.

[Wu 88] Wu, Y.-H.
Reduction: A Practical Mechanism of Searching for Regularity in Data.
In John Laird (editor), *Proc. of the Fifth International Conference on Machine Learning*, pages 374-380. University of Michigan, Ann Arbor, June, 1988.

Centre for Systems Science

• Simon Fraser University established the **Centre for Systems Scierce** as a provincial centre for advanced study. The objective of CSS is twofold: coordinate and combine advanced systems research at SFU, and encourage application of research results within industry in order to further the technological development of British Columbia.

•CSS has assembled three research groups to streamline its efforts. The orientation of these groups is as follows: intelligent systems, microelectronics, and computer and communications systems. Because of shared research interests, this latter group merges with LCCR (**Laboratory for Computer and Communications Research**) for the publication of technical reports.

• CSS's three groups involve over 80 skilled faculty researchers principally from SFU's Schools of Engineering Science and Computing Science, but also draw substantially on Mathematics, Physics, Geography, Philosophy, Linguistics, Psychology, Communication, and Kinesiology.